Лабораторная работа №1.

Julia. Установка и настройка. Основные принципы.

Ишанова А.И. группа НФИ-02-19

Содержание

List of Figures

1 Цель работы

Подготовить рабочее пространство и инструментарий для работы с языком программирования Julia, на простейших примерах познакомиться с основами синтаксиса Julia.

2 Задание

1. Установите под свою операционную систему Julia, Jupyter.
2. Используя JupyterLab, повторите примеры из раздела 1.3.3.
3. Выполните задания для самостоятельной работы.[1]

3 Теоретическое введение

Julia — высокоуровневый высокопроизводительный свободный язык программирования с динамической типизацией, созданный для математических вычислений. Эффективен также и для написания программ общего назначения. Синтаксис языка схож с синтаксисом других математических языков (например, MATLAB и Octave), однако имеет некоторые существенные отличия. Julia написан на Си, C++ и Scheme. Имеет встроенную поддержку многопоточности и распределённых вычислений, реализованные в том числе в стандартных конструкциях. [2]

Язык является динамическим, при этом поддерживает JIT-компиляцию (JIT-компилятор на основе LLVM входит в стандартный комплект), благодаря чему, по утверждению авторов языка, приложения, полностью написанные на языке (без использование низкоуровневых библиотек и векторных операций) практически не уступают в производительности приложениям, написанным на статически компилируемых языках, таких как Си или C++. Большая часть стандартной библиотеки языка написана на нём же. [2]

4 Выполнение лабораторной работы

4.1 Установка Julia и Jupyter

В ходе выполнения других курсов, мною уже была установлена Julia с IJulia для работы с Jupyter Notebook, поэтому этот пункт задания был мною опущен.

4.2 Повторение примеров

1. Повторяем примеры с определением типа числовой величины. (fig. 4.1)

```
[1]: typeof(3)
[1]: Int64
[2]: typeof(3.5)
[2]: Float64
[3]: typeof(3/3.5)
[3]: Float64
[4]: typeof(sqrt(3+4im))
[4]: ComplexF64 (alias for Complex{Float64})
[5]: typeof(pi)
[5]: Irrational{:π}
[15]: 1.0/0.0, 1.0/(-0.0), 0.0/0.0
[15]: (Inf, -Inf, NaN)
[16]: typeof(1.0/0.0), typeof(1.0/(-0.0)), typeof(0.0/0.0)
[16]: (Float64, Float64, Float64)
[6]: for T in
         [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
         println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
     end
    Int8: [-128,127]
   Int16: [-32768,32767]
   Int32: [-2147483648,2147483647]
   Int64: [-9223372036854775808,9223372036854775807]
  Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
   UInt8: [0,255]
  UInt16: [0,65535]
  UInt32: [0,4294967295]
  UInt64: [0,18446744073709551615]
 UInt128: [0,340282366920938463463374607431768211455]
```

Figure 4.1: Примеры определения типа числовых величин

2. Потворяем примеры приведения аргументов к одному типу. (fig. 4.2)

```
[7]: Int64(2.0)
[7]: 2
[8]: Char(2)
[8]: '\x02': ASCII/Unicode U+0002 (category Cc: Other, control)
[9]: convert(Int64, 2.0)
[9]: 2
[10]: convert(Char, 2)
[10]: '\x02': ASCII/Unicode U+0002 (category Cc: Other, control)
[11]: Bool(1)
[11]: true
[13]: Bool(0)
[13]: false
[14]: promote(Int8(1), Float16(4.5), Float32(4.1))
[14]: (1.0f0, 4.5f0, 4.1f0)
[17]: typeof(promote(Int8(1), Float16(4.5), Float32(4.1)))
[17]: Tuple{Float32, Float32, Float32}
```

Figure 4.2: Примеры приведения аргументов к одному типу

3. Повторяем примеры определения функций. (fig. 4.3)

```
[18]: function f(x)
          x^2
      end
[18]: f (generic function with 1 method)
[19]: f(4)
[19]: 16
[20]: g(x)=x^2
[20]: g (generic function with 1 method)
[21]: g(8)
[21]: 64
```

Figure 4.3: Примеры определения функций

1. Повторяем примеры работы с массивами. (fig. 4.4)

```
[22]: a = [4 7 6] # вектор-строка
      b = [1, 2, 3] # вектор-столбец
      a[2], b[2] # вторые элементы векторов a и b

[22]: (7, 2)

[23]: a = 1; b = 2; c = 3; d = 4 # присвоение значений
      Am=[a b; c d]#матрица2x2

[23]: 2×2 Matrix{Int64}:
       1  2
       3  4

[24]: Am[1,1], Am[1,2], Am[2,1], Am[2,2] # элементы матрицы

[24]: (1, 2, 3, 4)

[25]: aa = [1 2]
      AA = [1 2; 3 4]
      aa*AA*aa'

[25]: 1×1 Matrix{Int64}:
       27

[26]: aa, AA, aa'

[26]: ([1 2], [1 2; 3 4], [1; 2;;])
```

Figure 4.4: Примеры работы с массивами

4.3 Задания для самостоятельной работы

1. Изучили документацию по основным функциям Julia для чтения / записи / вывода информации на экран: read(), readline(), readlines(), readdlm(), print(), println(), show(), write(). Привели свои примеры их использования. (fig. 4.5 - fig. 4.16)

```
[27]: ?read()

[27]: read(io::IO, T)
      Read a single value of type T from io, in canonical binary representation.

      Note that Julia does not convert the endianness for you. Use ntoh or ltoh for this purpose.

      read(io::IO, String)
      Read the entirety of io, as a String (see also readchomp).
```

Figure 4.5: Фрагмент документации по команде read()

```
[29]: io = IOBuffer("Good night");
      read(io, String)

[29]: "Good night"
```

Figure 4.6: Пример по команде read()

```
[30]: ?readline()

[30]: readline(io::IO=stdin; keep::Bool=false)
      readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to `stdin`). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with `'\n'` or `"\r\n"` or the end of an input stream. When `keep` is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When `keep` is true, they are returned as part of the line.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n"
);
       end
57

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
julia> print("Enter your name: ")
Enter your name:

julia> your_name = readline()
Logan
"Logan"
```

```
[34]: readline("text.txt")

[34]: "Sleep well"
```

Figure 4.7: Документация по команде readline() и пример использования

```
[35]: ?readlines()
```

```
[35]: readlines(io::IO=stdin; keep::Bool=false)
      readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings. See also `eachline` to iterate over the lines without reading them all at once.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n"
);
       end
57

julia> readlines("my_file.txt")
2-element Vector{String}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Vector{String}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

```
[36]: readlines("text.txt")
```

```
[36]: 2-element Vector{String}:
       "Sleep well"
       "Eat well"
```

Figure 4.8: Документация по команде readlines() и пример использования

```
[41]: using DelimitedFiles
```

```
[42]: ?readdlm()
```

```
[42]: readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`.

Figure 4.9: Фрагмент документации по команде readlm()

```
[43]: readdlm("text.txt", '\t', String, '\n')
```

```
[43]: 2×1 Matrix{String}:
       "Sleep well"
       "Eat well"
```

```
[44]: x = [1; 2; 3; 4];
      y = [5; 6; 7; 8];

      open("delim_file.txt", "w") do io
          writedlm(io, [x y])
      end

      readdlm("delim_file.txt", '\t', Int, '\n')
```

```
[44]: 4×2 Matrix{Int64}:
       1  5
       2  6
       3  7
       4  8
```

Figure 4.10: Примеры по команде readlm()

```
[45]: ?print()
```

```
[45]: print([io::IO], xs...)
```

Write to `io` (or to the default output stream `stdout` if `io` is not given) a canonical (un-decorated) text representation. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

`print` falls back to calling `show`, so most types should just define `show`. Define `print` if your type has a separate "plain" representation. For example, `show` displays strings with quotes, and `print` displays strings without quotes.

See also `println`, `string`, `printstyled`.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello", ' ', :World!)

julia> String(take!(io))
"Hello World!"
```

```
[46]: print("Hi <3")
```

```
Hi <3
```

Figure 4.11: Документация по команде print() и пример использования

```
[47]: ?println()
```

```
println([io::IO], xs...)
```

Print (using `print`) `xs` to `io` followed by a newline. If `io` is not supplied, prints to the default output stream `stdout`.

See also `printstyled` to add colors etc.

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> String(take!(io))
"Hello, world.\n"
```

```
[48]: println("How are you?", "Good")
```

```
How are you?Good
```

Figure 4.12: Документация по команде println() и пример использования

```
[49]: ?show()
```

```
show([io::IO = stdout], x)
```

Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

To customize human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` instead. Checking the `:compact` `IOContext` property of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Figure 4.13: Фрагмент документации по команде show()

```
[50]: show("<3")
```

```
"<3"
```

Figure 4.14: Пример по команде show()

```
[51]: ?write()
```

```
[51]: write(io::IO, x)
      write(filename::AbstractString, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Return the number of bytes written into the stream. See also `print` to write a text representation (with an encoding that may depend upon `io`).

The endianness of the written value depends on the endianness of the host system. Convert to/from a fixed endianness when writing/reading (e.g. using `htol` and `ltoh`) to get results that are consistent across platforms.

You can write multiple values with the same `write` call. i.e. the following are equivalent:

```
write(io, x, y...)
write(io, x) + write(io, y...)
```

Figure 4.15: Фрагмент документации по команде write()

```
[52]: open("text2.txt", "w") do io
          write(io, "qwerty")
      end

      readlines("text2.txt")
```

```
[52]: 1-element Vector{String}:
       "qwerty"
```

Figure 4.16: Пример по команде write()

1. Изучили документацию по функции parse(). Привели свой пример её использования. (fig. 4.17)

```
[53]: ?parse()
```

```
[53]: parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

!!! compat "Julia 1.1" `parse(Bool, str)` requires at least Julia 1.1.

Examples

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

```
parse(::Type{Platform}, triplet::AbstractString)
```

Parses a string platform triplet back into a `Platform` object.

```
[54]: parse(Int64, "8")
```

```
[54]: 8
```

```
[55]: typeof(parse(Int64, "8"))
```

```
[55]: Int64
```

```
[56]: parse(Int, "f", base=16)
```

```
[56]: 15
```

Figure 4.17: Документация по команде parse() и пример использования

3. Изучили синтаксис Julia для базовых математических операций с разным типом переменных: сложение, вычитание, умножение, деление, возведение в степень, извлечение корня, сравнение, логические операции. Привели свои примеры. (fig. 4.18 - fig. 4.22)

```
[62]: #int

1+2, 1-2, 1*2, 1/2, 2^2, sqrt(4), 1>2, 1<2, 1==2, 1>=2, 1<=2, 1!=2, (1>2) && (2<3), (1>2) || (2<3)
```

```
[62]: (3, -1, 2, 0.5, 4, 2.0, false, true, false, false, true, true, false, true)
```

Figure 4.18: Примеры операций с типом int

```
[63]: #float
1.0+2.1, 1.1-2.1, 1.1*2.1, 1.1/2.1, 2.1^2, sqrt(4.1), 1.1>2.1, 1.1<2.1, 1.1==2.1, 1.1>=2.1,
1.1<=2.1, 1.1!=2.1, (1.1>2.1) && (2.1<3.1), (1.1>2.1) || (2.1<3.1)

[63]: (3.1, -1.0, 2.3100000000000005, 0.5238095238095238, 4.41, 2.0248456731316584, false, true,
false, false, true, true, false, true)
```

Figure 4.19: Примеры операций с типом float

```
[66]: #complex
1+2im, 1+2im-1im, 1im*1im, (1+2im)/2, (1im)^2, sqrt(1+2im), 1+2im==5+1im, 1+2im!=5+1im, (1+2im==5+1im) && (1im==1im),
(1+2im==5+1im) || (1im==1im)

[66]: (1 + 2im, 1 + 1im, -1 + 0im, 0.5 + 1.0im, -1 + 0im, 1.272019649514069 + 0.7861513777574233im, false, true, false, tru
e)
```

Figure 4.20: Примеры операций с типом complex

```
[68]: #rational
1//2+1//3, 1//2-1//3, 1//2*1//3, 1//2 / 3, (1//2)^2, 1//2 < 1//3, 1//2 <= 1//3, 1//2 > 1//3,
1//2 >= 1//3, 1//2 == 1//3, 1//2 != 1//3, (1//2>1//3) && (1//4 < 1//5), (1//2>1//3) || (1//4 < 1//5)

[68]: (5//6, 1//6, 1//6, 1//6, 1//4, false, false, true, true, false, true, false, true)
```

Figure 4.21: Примеры операций с типом rational

```
[75]: #irrational
using IrrationalExpressions

pi+2*pi, pi-2*pi, pi*pi, pi /2, sqrt(pi), pi < 2*pi, pi <= 2*pi, pi > 2*pi, pi >= 2*pi, pi == 2*pi,
pi != 2*pi, (pi < 2*pi) && (pi < pi/7), (pi < 2*pi) || (pi < pi/7)

[75]: (π + 2π = 9.424777960769..., π - 2π = -3.1415926535897..., π * π = 9.8696044010893..., π / 2 = 1.5707
9632679489..., 1.7724538509055159, true, true, false, false, false, true, false, true)
```

Figure 4.22: Примеры операций с типом irrational

4. Привели несколько своих примеров с операциями над матрицами и векторами: сложение, вычитание, скалярное произведение, транспонирование, умножение на скаляр. (fig. 4.23, fig. 4.24)

```
[79]: #векторы

      a = [1; 2]
      b = [3; 4]

      using LinearAlgebra

      a .+ b, a .- b, dot(a,b), a ⋅ b, a', a*2
```

```
[79]: ([4, 6], [-2, -2], 11, 11, [1 2], [2, 4])
```

Figure 4.23: Примеры операций с векторами

```
[81]: #матрицы

      A = [1 2; 3 4]
      B = [3 1; 2 4]

      A + B, A - B, dot(A,B), A', A*2
```

```
[81]: ([4 3; 5 8], [-2 1; 1 0], 27, [1 3; 2 4], [2 4; 6 8])
```

Figure 4.24: Примеры операций с матрицами

5 Листинг

```
# -*- coding: utf-8 -*-
# ---
# jupyter:
#   jupytext:
#     text_representation:
#       extension: .jl
#       format_name: light
#       format_version: '1.5'
#       jupytext_version: 1.14.1
#   kernelspec:
#     display_name: Julia 1.8.2
#     language: julia
#     name: julia-1.8
# ---

typeof(3)

typeof(3.5)

typeof(3/3.5)

typeof(sqrt(3+4im))
```

```
typeof(pi)
```

```
1.0/0.0, 1.0/(-0.0), 0.0/0.0
```

```
typeof(1.0/0.0), typeof(1.0/(-0.0)), typeof(0.0/0.0)
```

```
for T in
    [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
    println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
```

```
Int64(2.0)
```

```
Char(2)
```

```
convert(Int64, 2.0)
```

```
convert(Char, 2)
```

```
Bool(1)
```

```
Bool(0)
```

```
promote(Int8(1), Float16(4.5), Float32(4.1))
```

```
typeof(promote(Int8(1), Float16(4.5), Float32(4.1)))
```

```
function f(x)
```

```
    x^2
end

f(4)

g(x)=x^2

g(8)

a = [4 7 6] # вектор-строка
b = [1, 2, 3] # вектор-столбец
a[2], b[2] # вторые элементы векторов a и b

a = 1; b = 2; c = 3; d = 4 # присвоение значений
Am=[a b; c d]#матрица2x2

Am[1,1], Am[1,2], Am[2,1], Am[2,2] # элементы матрицы

aa = [1 2]
AA = [1 2; 3 4]
aa*AA*aa'

aa, AA, aa'

?read()

io = IOBuffer("Good night");
read(io, String)
```

```
?readline()

readline("text.txt")

?readlines()

readlines("text.txt")

using DelimitedFiles

?readdlm()

readdlm("text.txt", '\t', String, '\n')

# +
x = [1; 2; 3; 4];
y = [5; 6; 7; 8];

open("delim_file.txt", "w") do io
    writedlm(io, [x y])
end

readdlm("delim_file.txt", '\t', Int, '\n')
# -

?print()

print("Hi <3")
```

```
?println()

println("How are you?", "Good")

?show()

show("<3")

?write()

# +
open("text2.txt", "w") do io
    write(io, "qwerty")
end

readlines("text2.txt")
# -

?parse()

parse(Int64, "8")

typeof(parse(Int64, "8"))

parse(Int, "f", base=16)

# + tags=[]
#int
```

```
1+2, 1-2, 1*2, 1/2, 2^2, sqrt(4), 1>2, 1<2, 1==2, 1>=2, 1<=2, 1!=2, (1>2) && (2<3
# -

#float
1.0+2.1, 1.1-2.1, 1.1*2.1, 1.1/2.1, 2.1^2, sqrt(4.1), 1.1>2.1, 1.1<2.1, 1.1==2.1,
1.1<=2.1, 1.1!=2.1, (1.1>2.1) && (2.1<3.1), (1.1>2.1) || (2.1<3.1)

#complex
1+2im, 1+2im-1im, 1im*1im, (1+2im)/2, (1im)^2, sqrt(1+2im), 1+2im==5+1im, 1+2im!=
(1+2im==5+1im) || (1im==1im)

#rational
1//2+1//3, 1//2-1//3, 1//2*1//3, 1//2 / 3, (1//2)^2, 1//2 < 1//3, 1//2 <= 1//3, 1
1//2 >= 1//3, 1//2 == 1//3, 1//2 != 1//3, (1//2>1//3) && (1//4 < 1//5), (1//2>1//

# +
#irrational
using IrrationalExpressions

pi+2*pi, pi-2*pi, pi*pi, pi /2, sqrt(pi), pi < 2*pi, pi <= 2*pi, pi > 2*pi, pi >=
pi != 2*pi, (pi < 2*pi) && (pi < pi/7), (pi < 2*pi) || (pi < pi/7)

# +
#векторы

a = [1; 2]
b = [3; 4]

using LinearAlgebra
```

```
a .+ b, a .- b, dot(a,b), a ⊠ b, a', a*2

# +
#матрицы

A = [1 2; 3 4]
B = [3 1; 2 4]

A + B, A - B, dot(A,B), A', A*2
# -
```

6 Вывод

В ходе выполнения лабораторной работы на примерах были изучены основы синтаксиса в Julia.

7 Библиография

1. Методические материалы курса.
2. Wikipedia: Julia (язык программирования). (https://ru.wikipedia.org/wiki/Julia_(%D1%8F%D0